Micro-architectural Support for Metadata Coherence in Multi-core Dynamic Information Flow Tracking

Juan Carlos Martínez Santos*
Northeaster University
Boston, Massachusetts
jcmartin@ece.neu.edu

Yunsi Fei
Northeaster University
Boston, Massachusetts
yfei@ece.neu.edu

ABSTRACT

Dynamic information flow tracking (DIFT) has shown to be an effective security measure for detecting both memory corruption attacks and semantic attacks at run-time on a wild range of systems from embedded systems and mobile devices to cloud computing. When applying DIFT to multi-thread applications running on multi-core architectures, the data processing and metadata processing are normally decoupled, i.e., being performed in different places at different times. Therefore, if the metadata access is not in the same order as data access, inconsistency issues may arise, which would reduce the security effectiveness of DIFT. Avoiding such inconsistency between data access and metadata access, i.e., maintaining metadata coherence, has become a challenging issue. In this paper, we propose METACE (METAdata Coherence Enforcement). METACE includes architectural enhancement in the memory management unit and leverages the existing cache coherence hardware and protocol to enforce metadata coherence. It introduces minimum changes to cores, coprocessors, and the memory hierarchy. It covers the complete set of data dependencies without deadlocks and is compatible with different memory consistency models. Our approach does not require modification of the source code. METACE supports out-of-order metadata access resulting in less performance degradation than previous approaches.

Categories and Subject Descriptors

B.3.2 [**Memory Structures, Design Styles**]: Cache memories; D.3.4 [**Programming Languages, Processor, Compilers**]: Memory management; K.6.5 [**Computing Milieux**]: Security and Protection

Keywords

Cache Coherence; Metadata Coherence; Dynamic Information Flow Tracking; Software security

1. INTRODUCTION

*Currently on leave from Universidad Tecnológica de Bolívar, Cartagena, Colombia.

HASP'13 June 24, Tel-Aviv, Israel.

Dynamic information flow tracking (DIFT) has been used to monitor the run-time behavior of applications, as well as to defend effectively against both high-level semantic attacks [6] and low-level memory corruption attacks [23, 11, 5, 15, 20, 2]. The basic idea is to taint data inputs from untrusted sources with a tag (also called metadata), like network data and keyboard input, propagate the tag while data is being processed in the processor, and check the usage of tainted data in program's important sites, such as control-flow transfers and system calls. Metadata processing and its synchronization with data processing is an important factor that affects both the security effectiveness and the performance of DIFT implementations.

When multi-threaded programs run on multi-core systems with shared memory, *coherence* issues may arise with decoupled metadata processing. Figure 1 shows an example of consistent data and metadata access. Each application core (processing data) is associated with a coprocessor, which processes metadata. Memory-accessing events are shown on the individual time line of the main core or coprocessor. Data accessing order is defined by the programmer and ensured by the compiler. Decoupled metadata processing on a coprocessor is always later than the corresponding data processing on a main core, ensured either at static-time by the compiler or at run-time by the inter-core communications through first-in-first-out (FIFO) buffers or pipes. However, since the metadata processing on *coprocessor 1* is independent of that on *coprocessor 2*, event t_{41} may appear earlier than t_{32}. Therefore, *coprocessor 2* might get a stale value of $tag(U)$. This incorrect metadata value would affect the effectiveness of DIFT significantly, incurring false alarms or false negatives.

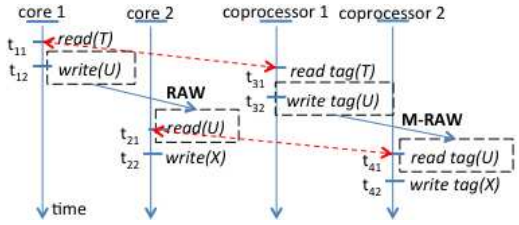


Figure 1: An example of consistent data access and metadata access. DIFT mechanism ensures that metadata processing is later than data processing ($t_{31} > t_{11}$, $t_{41} > t_{31}$)

In this paper, we propose METACE (METAdata Coherence Enforcement), a light-weight centralized micro-architectural approach, to keep data and metadata consistency in multi-core systems with shared memory and metadata processing decoupled on coproces-

sors. We slightly modify the existing cache coherence hardware to ensure that all data dependencies (read-after-write, write-after-read, and write-after-write) are preserved on the corresponding metadata (that we define as M-RAW, M-WAR, and M-WAW). Our solution is unintrusive to cores or coprocessors with minor changes in the on-chip shared interconnections. It can work with both in-order and out-of-order program execution, and is compatible with various memory consistency models (sequential, relaxed). We evaluate METACE on PARSEC programs, and the results show that it incurs small execution overhead.

The rest of the paper is structured as follows. Section 2 reviews the related work on metadata processing for DIFT. Section 3 describes our approach in details. Section 4 explains how METACE handles different critical data dependencies. Section 5 presents the experimental results, followed by further discussions in Section 6. Finally, Section 7 concludes the paper.

2. RELATED WORK

DIFT tagging techniques can be classified into two categories. The first type is "in-core" metadata processing, where the tag processing is integrated in the core pipeline. This requires drastic changes to both the processor logic and the register file, caches, memories, and busses for tag processing, storage, and propagating [23, 5, 6]. Such invasive implementations impose high design cost and incur hardware overhead. The second type is "decoupled" processing, where the metadata processing is decoupled from data processing to reduce the implementation complexity.

There are two types of "decoupled" metadata processing approaches, *offloading* DIFT on multi-cores [27, 14, 21, 16, 19, 24, 18], and *off-core* DIFT on coprocessors [10, 7, 9]. In *offloading* approaches, the original application extended with communication hooks runs on one core, and a replica with DIFT monitoring runs on another core [21, 16, 24]. In [19, 18], the extra core monitors the program execution without running the application. These techniques use shared memory to synchronize the data core and the monitor core. They are not intended to handle multi-thread applications with shared data except for [27], which uses hardware support to make the order of metadata updates sequentially consistent with data access adding a significant slowdown (40%) due to synchronization.

Off-core approaches attach a coprocessor to the main core and let the coprocessor monitor the application at run-time. A hardware interface (e.g., FIFO) handles the communication and synchronization between the main core and the coprocessor. *Off-core* approaches are applied to both single-thread applications [10, 7] as well as multi-thread [9]. In [9], the main core captures inter-thread data dependencies by monitoring the cache coherence traffic and enforces them on metadata processing on coprocessors. The added complexity to the coherence mechanism difficults its scalability.

To guarantee consistency between data and metadata, many works enforce atomicity by encapsulating both data and metadata accesses [13, 14]. Chung et al use transactional memory for metadata and data atomicity [4]. However, wrapping data and metadata access in a transaction requires costly hardware support for transactional memory access. Software approaches, like LIFT [17], instruments the application by inserting metadata operations after the corresponding data accesses. The execution overhead caused by atomicity and the instrumentation reduce its usability.

Our approach targets multi-thread applications running on multi-core architectures. We use *off-core* DIFT architecture, while the approach also applies to *off-loading* approaches with different performance impact. The main idea of METACE is to log memory accesses, identify data dependencies, and preserve them on corresponding metadata. METACE reduces the stall time of metadata processing by allowing out-of-order metadata processing on independent requests, rather than enforcing strict sequential data access order on all the metadata, as in [27]. METACE utilizes the current cache coherence protocol to monitor data memory accesses as in [9]. The distinct difference between both approaches is that our monitor hardware is centralized while the one in [9] is distributed. METACE enhances the Coherence Unit[1] instead of modifying each cache controller distributively. Our approach has a lower traffic in the shared bus. Our centralized structure shows a linear scalability instead of the exponential scalability shown in [9]. Overall, METACE is more resource-efficient and incurs less execution overhead.

3. OUR METACE APPROACH

3.1 DIFT and Metadata Coherence Overview

Our approach assumes *off-core* metadata processing similar to [10]. The application is being executed on main cores, and each main core is associated with a coprocessor for metadata processing. The main core notifies its coprocessor with important information about the instructions in execution through an FIFO buffer, as shows Figure 2. The coprocessor propagates the metadata and checks the usage of metadata in critical sites.

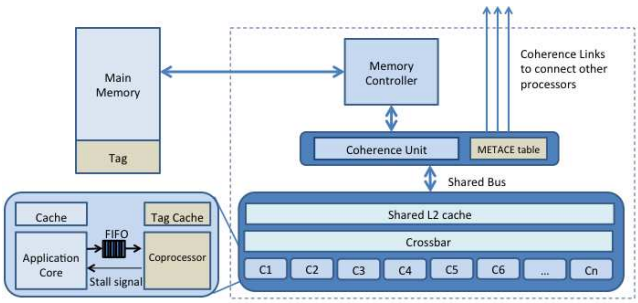


Figure 2: Our architecture enhancement for metadata coherence enforcement. TLBs and other on-chip details are omitted. This architecture is similar to the SPARC-T3 [22].

For different instructions running on the core, there are different ways of synchronization between the core and the coprocessor. For data processing like ALU execution instructions or memory access instructions, the metadata is processed by the coprocessor only after the instruction has been committed by the application core. For control instructions like system calls and indirect jumps, the core suspends after issuing the instruction and resumes execution only after the coprocessor examines that the control transfer actions abide by the security policy, i.e., the metadata of the control transfer target address is trusted.

3.2 Architecture Enhancement

METACE involves both architectural augmentation and cache coherence protocol extension. The current elaborate full cache coherency protocol is MOESI [12]. We modify the on-chip *Coher-*

[1]*Coherence Unit* (CU) is an on-chip module connected to the shared bus and the memory controller. It monitors all cache tags and controls cache coherence by transferring data between caches or between cache and the external memory [25]. CU is currently present in modern processors like SPARC T3 [22].

ence Unit by adding a table to track all data access events on the shared bus from applications cores. Each memory event adds an entry to the table, with the ID for the core requesting data access, the data address, and the action executed (read or write).

The coherence unit (CU) already snoops on the shared bus and controls the cache line status according to the MOESI protocol. METACE leverages the CU to fill up the new structure without much overhead. Metadata coherence is then enforced according to the table.

Figure 2 shows the architectural enhancement of METACE. The table resides in the $Coherence\ Unit$ and is associated with the shared bus in a multi-core architecture. For a multi-processor architecture, there are multiple CUs and therefore multiple tables, which are connected to other processors (CUs) via $Coherence\ Links$.

3.3 E-MOESI Protocol Implementation

We examine the original cache coherence protocol, MOESI, and find that it does not expose all the data access requests from main cores to the shared bus. We slightly augment the cache coherence protocol to capture the required information. Figure 3 shows the finite state machine for the extended MOESI cache coherence protocol (E-MOESI). Our modifications, shown in bold font, allow all the requests to be seen by the $Coherence\ Unit$ and the METACE table without changing the cache coherence functionality.

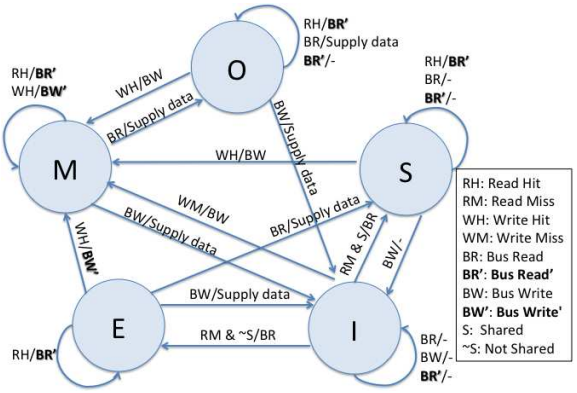


Figure 3: E-MOESI Cache Coherence Protocol. The text in bold indicates our modifications. $Bus\ Read'$ signal exposes read hit actions and $Bus\ Write'$ signal exposes write hit actions.

With the original MOESI, data access requests that result in a *cache hit* are not seen on the bus. We change the cache coherence protocol to broadcast a new signal, $Bus\ Read'$, on *cache hits*. In the case of $Read\ Hit$, the requesting core's cache line stays in its current status (M, O, E, or S), and the $Bus\ Read'$ signal is broadcasted into the bus. Similarly, on a $Write\ Hit$ of modified (M) or exclusive (E) data, $Bus\ Write$ signal is not generated under the original MOESI. Therefore, METACE table cannot see such requests. In our approach, the cache broadcasts a new signal, $Bus\ Write'$, in these cases. When the $Bus\ Write'$ and $Bus\ Read'$ signals are observed, no extra action is activated in other caches. This avoids additional bank/port contention on the other caches.

With these protocol changes, the cache coherence functionality is not affected at all, and all the data access requests are now seen by the $Coherence\ Unit$ and the METACE table. The extra signals on the shared bus would consume some bandwidth and affect the execution time. We have considered this overhead in our experiments.

3.4 Modeling Hardware Overhead of METACE

A naive in-order implementation of METACE works like an FIFO structure with a head pointer and a tail pointer, where an entry is appended to the FIFO's end each time a data access instruction is committed. Only the head entry can be evicted when a coprocessor's metadata access request matches it. The sequential access order of the FIFO structure determines that the first entry is blocking all the metadata accesses of other entries, similar to [27]. However, accessing metadata in exactly the same order as data access is not necessary and can stall non-critical requests, which introduces execution overhead on metadata processing. For example, for consecutive data readings by different cores, the order may not need to be preserved for metadata reading.

To support out-of-order metadata access and meanwhile maintain metadata coherence, a *fully-associative cache structure* is used for the table. Each entry is extended with two fields, $Key\ ID$ and $Lock\ ID$, which associate a set of entries with data dependencies among them. Within a set, there is only one entry holding the key, but there may be multiple entries with the same lock. Figure 4 shows the block diagram of the METACE table. The METACE logic is the controller that takes bus requests, manages the METACE table, and sends signals to the requesting main core or coprocessor.

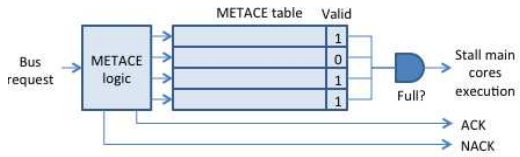


Figure 4: Block diagram of METACE.

Figure 5 depicts the actions executed inside the METACE modules when an entry is added. Each time a main core gains access to a memory data, a new entry should be added to the table. The METACE logic looks up the table for an available slot, and checks data dependency between it and the existing valid entries. If an RAW, WAR, or WAW dependency is discovered, a lock is set in the new entry and a key is assigned to the previous entry that must be attended first. For RAW, there may be multiple reads associated with one data write. Thus, the lock will be duplicated for the multiple read entries.

The *logging process* is activated by data access requests from main cores, and should be completed before the corresponding coprocessor requests metadata access. It is passively logging, and it is not on the execution path of neither main cores nor coprocessors. Even in basic 5-stage pipelines, the time window for the logging process (5 cycles) is long enough to accommodate the actions shown in Figure 5, which include table lookup, checking entry dependencies, and setting keys and locks. Therefore, memory event logging does not affect the program execution at all. Only when the table is full, a signal is generated to stall the execution on all main cores. When some table entries are evicted by metadata processing on coprocessors, the execution will resume.

Figure 6 shows the actions executed by the METACE modules when a coprocessor requests metadata access. The METACE table is looked up to find an entry with the corresponding data access that matches the request. If an entry is found and it is not locked, i.e., it does not have any data dependency to preserve, an acknowledg-

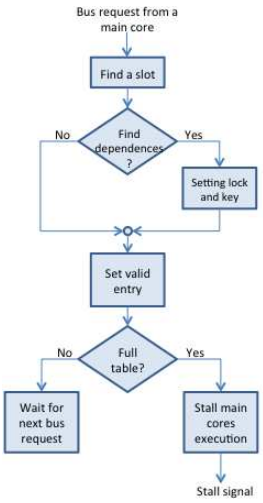


Figure 5: METACE actions when a data access entry from a main core is logged.

ment (ACK) signal is returned to the requester and the coprocessor gains access to the metadata instantly. If the matched entry holds a key for a lock, the corresponding lock is removed from other dependent entries. Then, the matched entry is deleted. The unlocked entries have no dependency any more and will be deleted when their corresponding metadata accesses are executed. If the matched entry is locked, the METACE logic rejects the metadata request by issuing an no-acknowledgment (NACK) signal to the request coprocessor. The coprocessor waits for other metadata access requests to unlock the entry. With such structure, any entry in the table can be evicted as long as certain conditions are met allowing out-of-order metadata processing.

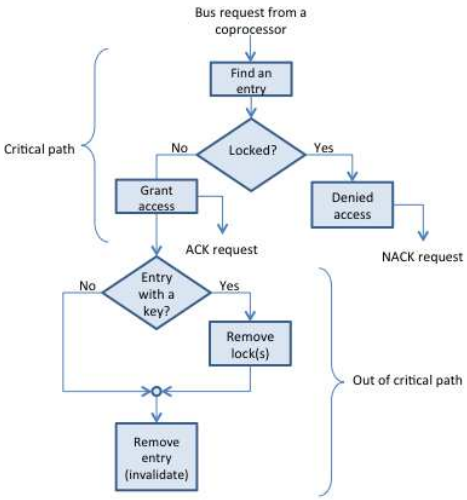


Figure 6: METACE actions when a coprocessor requests metadata access.

The METACE logic includes a fair bus arbitration mechanism that handles the shared communication bus. When a coprocessor is NACKed, the mechanism will schedule the coprocessor only when the dependent entry is resolved and removed from the table. As the METACE table is located in the on-chip coherence unit (CU), the access delay for looking up the table is equivalent to the access time to an on-chip translation looksaside buffer (TLB).

3.5 Overall Run-time Execution Overhead of METACE

Overall, there are some common execution overheads due to *off-core* DIFT techniques (shown in Section 3.1). Extra metadata accesses cause more memory accesses. The main application core may be stalled if the FIFO buffer between the core and the coprocessor is full, or the current instruction (critical control site) has to wait until the coprocessor checks if the instruction is allowed according to the DIFT security policy.

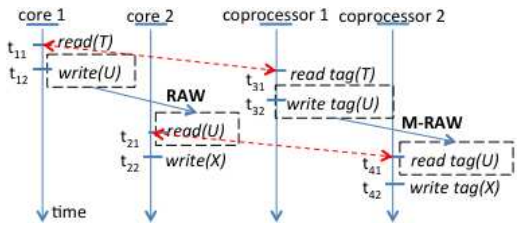
There are also METACE-specific factors that affect the performance, including increase of the traffic on the shared bus due to the enhanced cache coherence protocol (shown in Section 3.3), limited METACE table capacity which stall data processing when it is full and extra time to check the METACE table for access order (shown in Section 3.4).

In our experiments, we have considered all the aforementioned performance-affecting factors and analyzed their effects.

4. METADATA HAZARD ANALYSIS

4.1 Dealing with M-RAW, M-WAR, and M-WAW Hazards

To show how the METACE preserves RAW dependency on the corresponding metadata, we revisit the scenario presented in Figure 1, where two cores and two coprocessors execute the application and process the metadata in the correct sequence. Figure 7 shows the state of the table after memory access events are executed. Note that a lock is set on entry 3, and the key is assigned to entry 2. At the time $coprocessor\ 1$ requests to read $tag(T)$ (event t_{31} shown in Figure 1), the table is looked up for the request. Because the request (including the $core\ ID$, the $address$, and the action) matches the head entry and there is any lock, METACE allows $coprocessor\ 1$ to read the tag and the first entry is deleted from the list. Then, when $coprocessor\ 1$ requests to write $tag(U)$, the system allows the action, and the second entry is deleted from the list. Meanwhile, the lock #1 is removed from the dependent entry 3. When the $coprocessor\ 2$ requests read $tag(U)$, the METACE also allows the action.



Entry	Valid	Core ID	Address	Action	Key ID	Lock ID
1	1	1	T	Read		
2	1	1	U	Write	#1	
3	1	2	U	Read		#1
4	1	2	X	Write		

Figure 7: The state of METACE table after $core\ 1$ and $core\ 2$ access data in Figure 1, with a RAW between events t_{12} and t_{21}.

In the case of data-metadata inconsistency, i.e., $coprocessor\ 2$

requests to read $tag(U)$ (event t_{41}) earlier than $coprocessor$ 1 writes $tag(U)$ (event t_{32}), the table is looked up. There is an entry matching the request, entry 3 in Figure 7. However, the entry is locked, and therefore METACE logic sends a NACK signal to $coprocessor$ 2, which is taken as a cache miss signal. The metadata reading will be allowed only after $coprocessor$ 1 writes $tag(U)$ and unlocks entry 3.

To avoid M-RAW hazards, we set locks on read events by write events, but the system is still susceptible to M-WAR hazards. Similarly, read actions should also be logged and lock the dependent write if a WAR data dependency is found.

An M-WAW hazard will appear when two cores write to the same location without reading operations in between, and the compiler does not remove such dynamically dead code. This hazard will be resolved in the similar way as we handle M-RAW and M-WAR hazards.

4.2 Out-of-order Metadata Access

Figure 8 shows an example where two set of instructions have both WAR and RAW dependencies on two data (B and A), respectively. After the application cores execute the memory accesses, the table looks like what is shown in Figure 9. There are two data dependencies between $coprocessor$ 1 and $coprocessor$ 2, RAW on A (lock #1) and WAR on B (lock #2). Because there is no lock for entry 1 and 2, the $coprocessor$ 1 will not be stalled if it issues a request for reading $tag(C)$ (event t_{32}) earlier than reading $tag(B)$ (event t_{31}), i.e., out-of-order metadata access. Similarly, $coprocessor$ 2 can read $tag(C)$ (event t_{42}) any time because there is no lock (data dependency) on it.

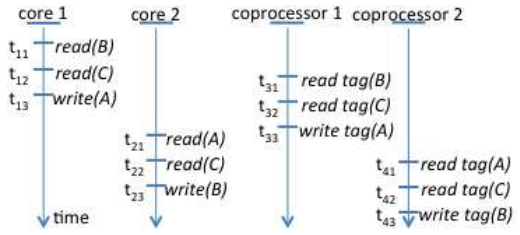


Figure 8: An example with WAR and RAW for out-of-order metadata access

Entry	Valid	Core ID	Address	Action	Key ID	Lock ID
1	1	1	B	Read	#2	
2	1	1	C	Read		
3	1	1	A	Write	#1	
4	1	2	A	Read		#1
5	1	2	C	Read		
6	1	2	B	Write		#2

Figure 9: The state of METACE table after main core data accesses in Figure 8 are executed

Overall, METACE keeps a log of all the data dependencies and enforces them on metadata access. It allows out-of-order metadata processing when no dependency is found.

5. EVALUATION

5.1 Experimental Setup

We use Multi2Sim [26], a cycle accurate multiprocessor simulator, to model and simulate different multi-core configurations and measure the performance impact of our approach. We extend the default cache module, *NMOESI*, to make all the data accesses visible to the centralized table.

Our baseline system has a two-level cache hierarchy similar to SPARC T3 [22]. The cache parameters has been set at the same values as in [9] to facilitate fair comparisons. The L1 instruction and data cache are 4-way 64KB each, the unified L2 cache is 4-way 32MB, and the tag cache is 4-way 8KB. All cache line sizes in the hierarchy are 64B. The LRU replacement policy is used for all caches. The access latencies for L1 instruction cache, data cache, and tag cache are all 3 cycles, and that for L2 cache is 6 cycles. The L2 cache miss penalty is 160 cycle. The goal of using a large L2 cache is to reduce the number of accesses to the main memory so that the effect of METACE on performance can be clearly exposed.

We model decoupled *off-core* DIFT monitors on coprocessors with a 16-entry FIFO interface, similar to [7, 10]. The amount of tags is proportional to the size of the program. We use: (i) 1-bit tag per byte, (ii) OR rule for tag propagation, and (iii) system calls and indirect jumps as the checking sites. The metadata is stored in the application's memory space. A translation table filled in at the program loading time is used to map the location for the metadata from the address of the corresponding data, as in [13].

The tag cache is similar to a regular data cache, except for manipulation at the bit-level granularity. A 32-bit enable mask is used for either reading a one-bit tag from the 32-bit retrieved word or updating one bit. As we set the same line size for both application core caches and coprocessor tag caches, a single metadata cache line can pack tags for eight data cache lines ideally, providing good spatial locality and therefore alleviating the performance slowdown. If one coprocessor attempts to access a metadata that is not altered, and it locates in a cache line with another dirty metadata, the cache coherence protocol requires to reload the whole line. This false sharing may increase the metadata cache miss rate. Details about how the architectural changes affect the execution performance are given in Section 5.3.

We choose a diverse set of CPU-intensive parallel benchmarks from the PARSEC Suite [1] to test the impact of our metadata coherence mechanism. We use the *simsmall* input set. To identify performance bottlenecks, the number of threads is equal to the number of cores running the application. The benchmarks are compiled for x86 architecture using $pthread$ synchronization primitives.

For each architecture configuration, we run the benchmarks on the multi-core system for three scenarios: 1) with coprocessors performing taint processing but without metadata coherence enforcement; 2) enforcing in-order metadata access with the additional hardware in the form of FIFO; 3) out-of-order metadata access with a fully-associative cache-like table.

5.2 Hardware Overhead for a Fully Associative METACE Table

Our experiments show that the size of the METACE table depends on the number of the thread running and not the size of the input set. With more threads, the METACE table is filled up more often, thereby either having a more severe performance impact due to stalling waiting on a METACE table entry to be freed up, or the implementation cost will be greater due to needing a large METACE table. By scaling the number of entries of the table to the number of cores, our simulations show that the selected sizes (16, 32, 64, 128 and 256 entries for 2, 4, 8, 16 and 32 cores, respectively) are sufficient to allow the application run without stalling the main application due to a full table. These values take in ac-

count the worst scenario found during the simulation stage, where each core was able to gain access to eight memory address before the corresponding coprocessors.

Table 1 shows the details of a table entry for a 32-core 32-bit architecture. Each entry is 64-bit. The first bit indicates the valid status of the entry. When deleting an entry, the valid bit is set to zero. The next three fields hold the data access information: the requesting core ID, data address, and access action. The key and lock identification numbers are each 8-bit, to support up to 256 dependencies. The last field is not used in the current implementation. It can be used in multi-process scenarios to label the entry ownership by different processes, so on context switch only a portion of the table is flushed rather than the entire table.

Valid	Core ID	Address	Action	Key ID	Lock ID	Unused
1 bit	5 bits	32 bits	1 bit	8 bits	8 bits	9 bits

Table 1: Details of an entry for an 8-core 32-bit architecture.

We use CACTI 5.3 [8] to estimate the on-chip area overhead of our METACE table, and compare it with the on-chip L1 cache under 65nm process technology. Table 2 shows the simulation results. In the worst scenario, the area overhead for a 256-entry table is around 0.12% of the die size of the SPART T3 Processor.

Area	32 entries	64 entries	128 entries
mm^2	0.044	0.070	0.152
Area	256 entries	L1 cache	SPARC T3 [22]
mm^2	0.463	2.335	371

Table 2: Comparison of area for different table sizes versus an L1 cache (64KB, 64B line, 4-way associativity) and a processor.

We count the total number of $Bus\ Read'$ and $Bus\ Write'$ signal broadcast in each configuration to have an idea about the power/energy impact of our approach. Figure 10 shows the additional activity in the shared bus and the tendency when the number of threads increases. In the worst case, the overhead is around 3.5%, which also show that the contention in the shared bus is low. We can see that the increase remains constant up to 8 cores. However, the tendency for more cores depends on the application. For $blackscholes$ and $canneal$ the tendency remains contant. For $fluidanimate$ and $swaptions$ the extra activity tends to decrease. For $streamcluster$ the abrupt change is due to the point dimensions (32) is the same number of cores, which stress the shared channel more than other configurations.

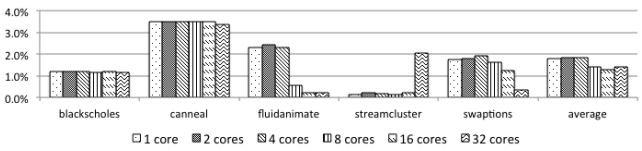


Figure 10: Increase of the shared bus utilization along with number of cores.

5.3 Performance Evaluations

We evaluate the impact of our approach on the execution time. Figure 11 shows the execution time overhead for the three cases, the application running with: (i) *off-core* DIFT monitoring only, denoted as DIFT, (ii) *off-core* DIFT monitoring and enforcing in-order metadata access, denoted as in-order, and (iii) *off-core* DIFT monitoring and allowing out-of-order metadata access, denoted as out-of-order, all normalized to the execution time for the application running without any DIFT. The system has 32 cores and 32 coprocessors, supporting 32-thread per application.

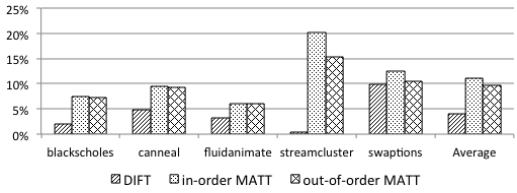


Figure 11: Execution time overhead of different DIFT implementations running 32 threads on 32 application cores, normalized to the execution time without DIFT monitoring.

We can see that our version of DIFT with out-of-order metadata coherence enforcement has a low performance degradation compared to application execution without DIFT, 9.7% on average. For the in-order processing, the execution time overhead is slightly higher, 12% on average. For applications with low and medium data sharing like $blackscholes$ and $fluidanimate$, the execution time overhead is lower because there are few data dependencies existing. For the other three applications, $canneal$ and $streamcluster$, with high or medium data exchange, and $swaptions$, with coarse data granularity, the execution time overhead is larger.

Adding metadata coherency enforcement to DIFT mechanism introduces very low overhead on top of off-core DIFT, less than 5.6% on average for our out-of-order metadata access. These performance results are similar to the 7% execution overhead over DIFT for a distributed approach [9]. The advantages of our approach are less invasive (without change the distributed cache controllers), easier and cheaper to scale with more cores, and can be easily extended for multi-processor architecture. In these cases, the execution overhead would increase linearly instead of exponentially as happens in [9].

Figure 12 shows the performance degradation of our out-of-order metadata processing implementation when the number of cores changes, normalized to DIFT without metadata coherence enforcement. The source of overhead is the extra traffic added to the shared bus to monitor data accesses and enforce metadata coherence. Overall, the execution time overhead is small, around 4% for 8 applications cores, and less than 6% for 16 and 32 cores application cores. With more cores requesting access to the bus, there are more bus contentions for cache coherency and metadata coherency.

For single-thread applications, there is no coherency issue. Therefore, the approach proposed in this paper should not be activated at all. If single-thread application is running, there will be power/energy wasted on extra traffic on the bus, but the program performance will not be affected much. Figure 12 shows that the performance overhead is 4%.

It is interesting to note that as the number of cores increases, the overhead is not always increasing. The more cores, the more contentions on the shared bus for metadata coherency enforcement, and therefore the absolute execution overhead is increasing. However, the baseline performance (DIFT without metadata coherence

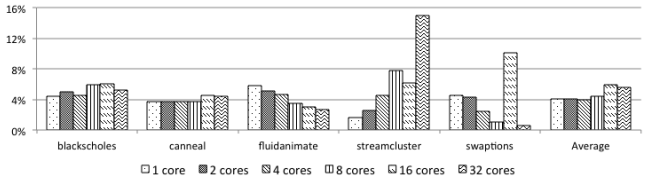


Figure 12: Execution time overhead with out-of-order access processing on 1, 2, 4, 8, 16, and 32 application cores and the same number of coprocessors, normalized to the DIFT execution time without metadata coherence enforcement.

enforcement) is not always improving along the number of cores, because the cache coherency mechanism may even increase DIFT's execution time when the number of cores increase. Therefore, the normalized overhead due to metadata coherence enforcement does not always deteriorate.

To investigate more details of the performance slowdown of METACE, we break down the overhead of enforcing metadata into three parts: *DIFT overhead*, *MOESI overhead*, and *enforcement overhead*. Figure 13 and Figure 14 show how it is composed for in-order and out-of-order implementation respectively. *DIFT overhead* represents the cost of decoupled metadata processing (initialization, propagation, and checking) on the *off-core* coprocessors. *MOESI overhead* is the extra overhead added to the system to make all the data accesses visible to the table (shared bus traffic). The last segment is the portion of the overhead for enforcing metadata coherence by looking up the table and checking the conditions. In Figure 13, *in-order overhead* is the extra overhead for enforce strict order metadata processing. In Figure 14, *out-of-order overhead* is the potion for allowing out-of-order metadata processing. Each application is running with 32 threads on 32 cores. As we can see, the out-of-order overhead is small, confirming our analysis in Section 3.5. The overhead for metadata processing (DIFT) and especially the coherence protocol enhancement (E-MOESI) are the biggest components of the overall overhead, indicating potentials for future performance optimizations.

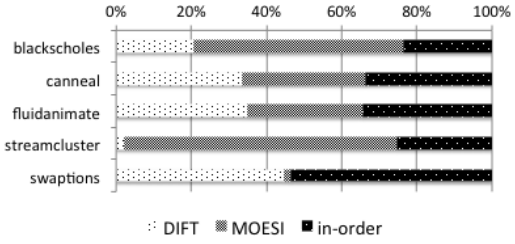


Figure 13: Overhead composition for in-order metadata processing. Each application is running with 32 threads on 32 cores.

6. DISCUSSIONS AND FUTURE WORK

Our approach requires a centralized *coherence unit* to make the work scalable. For architectures without coherence unit e.g., tiled processors or NUMA processors, a good solution should be the adoption of state-of-art memory race-recording schemes such as LReplay [3] and Timetraveler [28] in conjunction with the METACE table to expose and log all memory accesses.

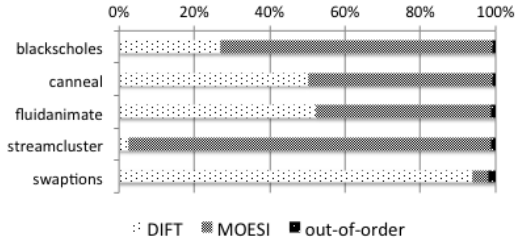


Figure 14: Overhead composition out-of-order metadata processing. Each application is running with 32 threads on 32 cores.

As future work, we plan to test our approach in different architectures like Nehalem, and evaluate the effect of our approach on the performance with different architecture features, including L3 cache, distributed Coherence Units, and multiple processors.

7. CONCLUSIONS

Decoupled metadata processing may result in data-metadata inconsistency issue, affecting both the security effectiveness and performance of DIFT implementations. This paper presents METACE, a centralized architectural enhancement in the coherence unit that enforces metadata coherence for dynamic information flow tracking in multi-thread applications running on multi-cores with shared memory. By monitoring the cache events, our approach identifies data dependencies and avoids data-metadata inconsistency. The extra hardware is unintrusive since it only introduces an extra module on the shared bus, rather than in the cores or coprocessors. Our simulation results show that the overhead for doing DIFT is dependent on the application's features of data sharing and data exchanging. The results have demonstrated that our centralized solution only slightly affects the overall execution time, and can be implemented with much lower complexity and higher resource efficiency than the distributed approach.

8. ACKNOWLEDGEMENTS

We would like to thank the anonymous HASP reviewers for their comments and feedback on the ideas in this paper. This work was supported by National Science Foundation under CAREER grant CNS-0845871.

9. REFERENCES

[1] C. Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[2] S. Chen, J. Xu, N. Nakka, Z. Kalbarczyk, and R. Iyer. Defeating memory corruption attacks via pointer taintedness detection. In *Proc. Int. Conf. on Dependable Systems & Networks*, pages 378 – 387, June 2005.

[3] Y. Chen, W. Hu, T. Chen, and R. Wu. LReplay: a pending period based deterministic replay scheme. *SIGARCH Comput. Archit. News*, 38(3):187–197, June 2010.

[4] J. Chung, M. Dalton, H. Kannan, and C. Kozyrakis. Thread-safe dynamic binary translation using transactional memory. In *Proc. IEEE Int. Symp. on High Performance Computer Architecture*, pages 279 –289, Feb. 2008.

[5] J. R. Crandall, S. F. Wu, and F. T. Chong. Minos: Architectural support for protecting control data. *ACM Tran. Architecture & Code Optimization*, 3(4):359–389, Dec. 2006.

[6] M. Dalton, H. Kannan, and C. Kozyrakis. Raksha: A flexible flow architecture for software security. In *Proc. Int. Symp. Computer Architecture*, pages 482–293, June 2007.

[7] D. Y. Deng, D. Lo, G. Malysa, S. Schneider, and G. E. Suh. Flexible and efficient instruction-grained run-time monitoring using on-chip reconfigurable fabric. In *Proc. IEEE/ACM Int. Symp. on Microarchitecture*, pages 137–148, Dec. 2010.

[8] HP Labs. CACTI 5.3. http://quid.hpl.hp.com:9081/cacti/.

[9] H. Kannan. Ordering decoupled metadata accesses in multiprocessors. In *Proc. Int. Symp. Microarchitecture*, pages 381 –390, Dec. 2009.

[10] H. Kannan, M. Dalton, and C. Kozyrakis. Decoupling dynamic information flow tracking with a dedicated coprocessor. In *Proc. Int. Conf. Dependable Systems & Networks*, pages 105 –114, Jun. 2009.

[11] J. C. Martinez Santos, Y. Fei, and Z. J. Shi. PIFT: Efficient dynamic information flow tracking using secure page allocation. In *Proc. Wkshp on Embedded Systems Security*, pages 6:1–6:8, Oct. 2009.

[12] MOESI Protocol. AMD64 Architecture Programmer's Manual: V2: System Programming. http://support.amd.com/us/Embedded_TechDocs /24593.pdf.

[13] V. Nagarajan and R. Gupta. Architectural support for shadow memory in multiprocessors. In *Proc. ACM SIGPLAN/SIGOPS Int. Conf. on Virtual Execution Environments*, pages 1–10, Mar. 2009.

[14] V. Nagarajan, H.-S. Kim, Y. Wu, and R. Gupta. Dynamic information flow tracking on multicores. In *Proc. Wkshp on Interaction between Compilers & Computer Archirectures*, 2008.

[15] J. Newsome. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Int. Symp. on Software Testing & Analysis*, Feb. 2005.

[16] E. B. Nightingale, D. Peek, P. M. Chen, and J. Flinn. Parallelizing security checks on commodity hardware. In *Proc. Int. Conf. Architectural Support for Programming Languages & Operating Systems*, pages 308–318, Mar. 2008.

[17] F. Qin, C. Wang, Z. Li, H. seop Kim, Y. Zhou, and Y. Wu. LIFT: A low-overhead practical information flow tracking system for detecting security attacks. In *IEEE/ACM Int. Symp. on Microarchitecture*, pages 135–148, Dec. 2006.

[18] O. Ruwase, S. Chen, P. B. Gibbons, and T. C. Mowry. Decoupled lifeguards: Enabling path optimizations for dynamic correctness checking tools. In *Proc.ACM SIGPLAN Conf. on Programming Language Design & Implementation*, pages 25–35, June 2010.

[19] O. Ruwase, P. B. Gibbons, T. C. Mowry, V. Ramachandran, S. Chen, M. Kozuch, and M. Ryan. Parallelizing dynamic information flow tracking. In *Proc. Symp. Parallelism in Algorithms & Architectures*, pages 35–45, June 2008.

[20] R. Shetty, M. Kharbutli, Y. Solihin, and M. Prvulovic. HeapMon: A helper-thread approach to programmable, automatic, and low-overhead memory bug detection. *IBM J. Res. Dev.*, 50:261–275, March 2006.

[21] W. Shi, H.-H. S. Lee, L. 'Falk, and M. Ghosh. An integrated framework for dependable and revivable architectures using multicore processors. In *Proc. Int. Symp. on Computer Architecture*, ISCA '06, pages 102–113, June 2006.

[22] SPARC T3-1, SPARC T3-2, SPARC T3-4 and SPARC T3-1B Server Architecture. Sun Oracle. http://www.oracle.com/technetwork/articles/systems-hardware-architecture/sparc-t3-server-architecture-176017.pdf, Febrary 2011.

[23] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *Proc. Int. Conf. on Architectural Support for Programming Languages & Operating Systems*, pages 85–96, 2004.

[24] M. Susskraut, S. Weigert, U. Schiffel, T. Knauth, M. Nowack, D. B. Brum, and C. Fetzer. Speculation for parallelizing runtime checks. In *Proc. Int. Symp. on Stabilization, Safety, & Security of Distributed Systems*, pages 698–710, Nov. 2009.

[25] M. Takahashi, H. Takano, E. Kaneko, and S. Suzuki. A shared-bus control mechanism and a cache coherence protocol for a high-performance on-chip multiprocessor. In *Proc. Int. Symp. on High-Performance Computer Architecture*, pages 314 –322, Feb. 1996.

[26] R. Ubal, J. Sahuquillo, S. Petit, and P. López. Multi2Sim: A Simulation Framework to Evaluate Multicore-Multithreaded Processors. In *Proc. Int. Symp. on Computer Architecture and High Performance Computing*, Oct. 2007.

[27] E. Vlachos, M. L. Goodstein, M. A. Kozuch, S. Chen, B. Falsafi, P. B. Gibbons, and T. C. Mowry. ParaLog: Enabling and accelerating online parallel monitoring of multithreaded applications. In *Proc. ACM on Architectural Support for Programming Languages & Operating Systems*, pages 271–284, Mar. 2010.

[28] G. Voskuilen, F. Ahmad, and T. N. Vijaykumar. Timetraveler: exploiting acyclic races for optimizing memory race recording. *SIGARCH Comput. Archit. News*, 38(3):198–209, June 2010.